บทความ **พิเศษ**

สหัส พรหมสิทธิ์

สารพัดมุมมองในการสร้างซอฟต์แวร์ให้มีคุณภาพ

องค์ประกอบที่มีส่วนในการทำให้ซอฟต์แวร์มีคุณภาพสูง พร้อมมุมมองในการเขียนโปรแกรมเพื่อสร้างซอฟต์แวร์เช่นในปัจจุบัน

ภาพที่กว้างที่สุดสำหรับในการสร้างคอมพิวเตอร์ซอฟต์แวร์ที่มีคุณภาพคงหนีไม่พ้นขอบเขตของงานที่เกี่ยวกับการออกแบบ การพัฒนาเพื่อให้ไปสู่เป้าหมายอันเป็นข้อต้องการ รวมทั้งการทดสอบซอฟต์แวร์โปรแกรมที่เขียนเสร็จแล้ว ถ้าหากจะมองให้ละเอียดลงไปอีกขั้น ก็จะเห็นว่าการออกแบบที่มีคุณภาพ (quality design) มักต้องคลุมถึงเรื่องของ

1. เทคโนโลยี (เช่นจะใช้ structured methods หรือ OO methods)

2. ขั้นตอนในการออกแบบ (design)

3. หลักในทางปฏิบัติเช่นในทางธุรกิจ (เช่นจะต้องนำเอาเรื่องของ บิสเนส-โปรเซส-รีเอ็นจิเนียริง เข้ามาพิจารณาด้วยหรือไม่)

รายละเอียดในการออกแบบยังสามารถขยายต่อไปได้ดังนี้คือ แผนและขอบเขต นิยามของธุรกิจ การวิเคราะห์และกระบวนการ รวมทั้งการสร้างต้นแบบสำหรับ GUI การแบ่งแยกประเภทของข้อมูลและกระบวนการ (partition data and processes) การเลือกเครื่องมือ (tools) ที่จะมาให้การสนับสนุนการพัฒนาซอฟต์แวร์ การสร้างฐานข้อมูล โปรแกรม เน็ตเวิร์ก และ GUI จนถึงขั้นของการทดสอบเพื่อให้ผู้ใช้ยอมรับ นอกจากนี้ก็ยังมีสิ่งที่ควรคำนึงถึงในแง่ที่ว่าในอดีตผู้เขียนโปรแกรมหรือสร้างซอฟต์แวร์จะต้องเข้าใจคอมพิวเตอร์เพียงไม่กี่ชนิด (เช่น เมนเฟรม) แต่ในปัจจุบัน ทุกอย่างได้เปลี่ยนไปมาก เช่นในการสร้างไคลเอนต์เซอร์ฟเวอร์แอพพลิเคชัน ซึ่งทำให้เกิดมีปัจจัยต่าง ๆ ให้ต้องนำมาพิจารณาอย่างมากมาย อันเป็นสิ่งที่เลี่ยงได้ยากสำหรับการใช้คอมพิวเตอร์ส่วนใหญ่ในขณะนี้

สร้างซอฟต์แวร์ให้มีคุณภาพ

องค์ประกอบที่มีส่วนในการทำให้ซอฟต์แวร์มีคุณภาพสูง น่าจะมีอยู่ 3 อย่าง นั่นก็คือ

1. คนหรือบุคลากร
2. เทคโนโลยี
3. กรรมวิธี หรือกระบวนการในการสร้างซอฟต์แวร์

สิ่งที่เป็นเรื่องใหญ่ที่สุดในที่นี้จึงหนีไม่พ้นเรื่องของคน (เช่นโปรแกรมเมอร์) โดยเฉพาะบุคคลที่มีคุณภาพ อันหมายถึงคุณภาพในด้านความรู้ในสิ่งตนต้องทำ จนถึงเรื่องของจรรยาบรรณความรับผิดชอบ ซึ่งถ้าจะให้สรุปก็คือว่าคุณภาพของซอฟต์แวร์จะแปรตามคุณภาพของคนหรือบุคลากรผู้เกี่ยวข้องกับการสร้างซอฟต์แวร์ ถัดไปก็เป็นเรื่องของเทคโนโลยี ซึ่งผู้พัฒนาซอฟต์แวร์ส่วนใหญ่มักให้ความสนใจเป็นพิเศษ ซึ่งก็น่าจะมีเหตุผลอยู่อย่างมาก เช่น ในกรณีที่กลุ่มของคอมพิวเตอร์ขนาดเล็กที่สุดในรูปของไคลเอนต์เซอร์ฟเวอร์ สามารถที่จะเอาชนะใจผู้ใช้เหนือระบบที่เป็นเมนเฟรม ดังจะเห็นได้จากเทคโนโลยีที่อยู่ในรูปของอินเทลลิเจนต์เวิร์กสเตชันและ GUIs ที่เปิดโอกาสให้ผู้ใช้คอมพิวเตอร์สามารถมีระบบที่มีคุณภาพที่สูงกว่าดัมเทอร์มินัลคาแร็กเตอร์เบสเมนเฟรม รวมทั้งเทคโนโลยีเช่น Repid Application Development (RAD) Environment Visual Basic, Delphi, PowerBuilder ฯลฯ ซึ่งได้เข้ามาทำให้โปรแกรมเมอร์สามารถสร้างระบบได้เร็วกว่าเมื่อครั้งที่ต้องใช้ COBOL และเมนเฟรม

อย่างไรก็ตาม ในท่ามกลางความสะดวกสบายดังกล่าวของโลกยุคใหม่ก็มักจะมีปัญหาตามมาเป็นการชดเชย เช่นในการที่ไคลเอนต์เซอร์ฟเวอร์เทคโนโลยีต้องเข้าไปเกี่ยวข้องกับระบบเปิด (open systems) อันทำให้ผู้พัฒนาซอฟต์แวร์ต้องเผชิญกับฮาร์ดแวร์แพลตฟอร์มที่อยู่ในรูปแบบต่าง ๆ ระบบดำเนินการที่มีให้เลือกใช้ได้มากมายหลายชนิด มิดเดิ้ลแวร์ DBMS-แพ็กเกจ, ภาษาโปรแกรมอันหลากหลาย และโปรโตคอลที่มีนับสิบ ซึ่งความหลากหลายดังกล่าวน่าจะทำงานอย่างเข้ากันได้ แต่ทุกอย่างก็ไม่ง่ายอย่างที่คิด ดังนั้นถ้าเกิดมีปัญหาข้อผิดพลาดขึ้นมาก็ไม่รู้จะไปโทษใคร นอกจากนี้เทคโนโลยีที่มีอานุภาพสูง (เช่น Visual Basic) ย่อมมีทั้งคุณและโทษ คือถ้าใช้ให้ถูกทางก็ดีไป แต่ถ้าใช้ไม่เป็นก็จะก่อให้เกิดความเสียหายได้เร็วขึ้น สุดท้ายก็เป็นเรื่องของการนำเทคโนโลยีไปให้การสนับสนุนกระบวนการ เช่น Prototyping Spiral Development หรือ RAD การให้ความสนใจในรายละเอียดของกระบวนการดังกล่าวจึงสำคัญมาก ที่สำคัญก็คือว่าเทคโนโลยีสามารถที่จะให้การสนับสนุนกระบวนการต่าง ๆ หรือทำให้เกิดความเป็นอัตโนมัติในกระบวนการดังกล่าว ตัวอย่างของผลิตภัณฑ์ที่ให้การสนับสนุนได้ทั้ง structured และ OO methodologies ก็มีอาทิเช่น System Architect, Paradign Plus ฯลฯ ทั้งหมดนี้ก็เป็นมุมมองอย่างกว้าง ๆ เพียงแง่มุมหนึ่งในโลกของการพัฒนาซอฟต์แวร์ในปัจจุบัน

formal methods

มุมมองที่สองในการเขียนโปรแกรมเพื่อสร้างซอฟต์แวร์เช่นในปัจจุบัน เป็นมุมมองในระดับจุลภาคที่สอดแทรกเข้ามาในแง่ของรายละเอียด ซึ่งต่างจากที่กล่าวในข้างต้น แต่ก็เป็นความต่างที่ส่งเสริมซึ่งกันและกัน เพราะเมื่อมองในมุมกว้าง (มหภาค) แล้ว ก็ควรหันไปมองมุมอื่นบ้างเพื่อความรอบรู้

หากใครสักคนที่อยู่ในเมืองใหญ่ ๆ แหงนหน้าขึ้นมองท้องฟ้า ก็มักจะพบว่าในเกือบจะทุกครั้งที่มองดูท้องฟ้าอันไม่ว่าจะเป็นในทิศทางไหน ก็จะเห็นเครื่องบิน (เช่นเครื่องบินโดยสาร) บินข้ามไปมาแทบไม่ว่างเว้น อย่างเช่นในกรุงลอนดอน (ต้องขออนุญาตใช้ตัวอย่างจากต่างประเทศ) ซึ่งได้มีการประมาณกันว่าในแต่ละขณะ จะมีเครื่องบินโดยสารไม่ต่ำกว่าห้าลำ (ตัวเลขปี 1994) ที่บินอยู่เหนือสนามบินขนาดใหญ่แห่งใดแห่งหนึ่ง (จากจำนวนสองแห่ง) เพื่อรอเวลาในการลงสู่สนามบิน รวมทั้งได้มีการคาดกันต่อไปอีกว่า ในปี ค.ศ. 2005 ผู้โดยสารทางอากาศของประเทศอังกฤษจะมีมากถึง 188 ล้านคน อันทำให้การควบคุมการจราจรทางอากาศในกรุงลอนดอน (รวมทั้งในเมืองใหญ่ ๆ แห่งอื่น ๆ ของโลก) เกิดเป็นปัญหาขึ้นมาในแง่ของการต้องการการปรับปรุงระบบควบคุมการจราจรทางอากาศใหม่หมด ดังเช่นการอัพเกรด London Area Terminal Control Center ในปี 1992 ซึ่งได้มีข้อต้องการที่เข้มงวดมากในเรื่องของระบบควบคุมฯ ในแง่ที่ว่า ระบบดังกล่าวจะต้องสามารถให้ข้อมูลที่เกี่ยวกับสภาพอากาศ สถานภาพของสนามบิน การมาถึงและการเตรียมตัวขึ้นสู่อวกาศของเครื่องบินโดยสาร แก่ผู้ที่ทำหน้าที่เป็น traffic managers ได้ในทุกวินาที โดยเพียงแต่กดปุ่มเท่านั้น ที่สำคัญก็คือว่า ข้อมูลต่าง ๆ ที่ปรากฏบนจอภาพของระบบควบคุมฯ จะต้องมีความละเอียดแม่นยำสูง เพื่อเป็นการสนองตอบในด้านความปลอดภัยของชีวิตของผู้โดยสารนับหมื่นในแต่ละวันที่ฝากชีวิตไว้กับความไว้ใจได้ของซอฟต์แวร์ที่มีความสำคัญในระดับ safety critical กล่าวคือ ถ้าซอฟต์แวร์เกิดพัง (crashes) เครื่องบินก็อาจพังตามเพราะอาจเกิดอุบัติเหตุในขณะบินขึ้นหรือลงสู่สนามบิน การออกแบบซอฟต์แวร์ประเภทนี้จึงต้องการความระมัดระวังและความสนใจในรายละเอียดอย่างยิ่งยวด อันเป็นเหตุให้ CAA (Civil Aviation Authority ของอังกฤษ) กับบริษัท Praxis อันเป็นที่ปรึกษาด้านซอฟต์แวร์ได้เลือกใช้เทคนิคในการสร้างซอฟต์แวร์ที่เรียกว่า formal methods ในระหว่างการออกแบบ สิ่งที่เรียกว่ากรรมวิธีหรือกระบวนการ (process) สำหรับระบบควบคุมฯ formal methods จึงแตกต่างจากวิธีธรรมดาโดยทั่วไปอย่างมาก เพราะอาศัยโมเดลทางคณิตศาสตร์ของภาษาซอฟต์แวร์ (mathematical models of software languages) แทนที่จะใช้ภาษาธรรมดา สำหรับสร้างข้อกำหนดรายละเอียดว่าซอฟต์แวร์ต้องทำอะไรบ้าง ที่สำคัญก็คือว่าผู้ออกแบบ process สามารถที่จะแก้สมการในโมเดลทางคณิตศาสตร์ เพื่อทดสอบผลของการออกแบบซอฟต์แวร์ของตน

ในทำนองเดียวกัน ในขณะที่ได้มีการนำไมโครโปรเซสเซอร์ (พร้อมด้วยซอฟต์แวร์ที่บรรจุอยู่ภายในนั้น) ไปใช้กับผลิตภัณฑ์หรือระบบที่เป็นแบบ safety critical มากขึ้น อันนับแต่จะมีความสำคัญมากขึ้นทุกที อย่างไรก็ตาม สถิติเท่าที่ปรากฏได้แสดงว่า โครงการออกแบบซอฟต์แวร์ต่าง ๆ มักไม่ประสบความสำเร็จเท่าใดนัก เพราะราวหนึ่งในสี่ของระบบซอฟต์แวร์ขนาดใหญ่ต้องถูกยกเลิกในระหว่างการพัฒนา และอีกราวสามในสี่ของระบบขนาดใหญ่ที่ได้พัฒนาจนแล้วเสร็จได้มีข้อผิดพลาดจนใช้การไม่ได้ หรือใช้การไม่ได้ดี (ข้อมูลจากวารสาร New Scientist ฉบับ 26 สิงหาคม 1995) อันน่าจะเป็นเหตุผลว่า ทำไมบริษัทที่ปรึกษาชั้นนำของอังกฤษเช่น Praxis จึงได้รับสัญญาว่าจ้างจากสำนักงานควบคุมการจราจรทางอากาศของอังกฤษ (CAA) ให้เป็นผู้ใช้ formal methods ในการออกแบบซอฟต์แวร์สำหรับระบบควบคุมฯ ดังได้กล่าวแต่ต้น บริษัท Praxis นั้นเชี่ยวชาญในด้านซอฟต์แวร์ที่เกือบจะเรียกได้ว่าเป็นชั้นนำของโลก เมื่อเป็นเช่นนี้จึงอยากจะนำข้อคิดเห็นของนายแอนโธนี ฮอลล์ ซึ่งเป็นที่ปรึกษาอาวุโสของบริษัทนี้มาเล่าให้ฟังว่า formal methods สามารถให้ความหวังในแง่ของการปรับปรุงและส่งเสริมประสิทธิภาพแก่กรรมวิธีในการออกแบบซอฟต์แวร์ที่ใช้กันอยู่ทั่วไป ในแง่ที่ว่าถ้านำ formal methods ไปใช้ร่วมกับภาคปฏิบัติการทางซอฟต์แวร์เอ็นจิเนียริงที่ดีก็จะทำให้ได้ซอฟต์แวร์ที่มีคุณภาพสูงขึ้นโดยไม่ต้องเพิ่มค่าใช้จ่าย ข้อได้เปรียบของการใช้ formal methods ร่วมกับวิธีอื่นก็คือว่าจะทำให้ผู้ออกแบบซอฟต์แวร์สามารถเข้าใจข้อต้องการต่าง ๆ ได้อย่างชัดเจน รวมทั้งให้ผลของการออกแบบที่เรียบง่ายและสามารถทดสอบได้อย่างสะดวก ที่สำคัญก็คือสามารถช่วยให้ผู้ออกแบบซอฟต์แวร์สามารถเผชิญกับส่วนที่มีความซับซ้อนมาก ๆ ได้ ทั้งหมดนี้ก็เป็นทัศนะของนายแอนโธนี ฮอลล์ จากบริษัท Praxis สำหรับหน่วยงานหรือองค์กรอื่น ๆ ในยุโรปที่มองเห็นความสำคัญของเรื่องนี้ก็มีเช่น The International Electrotechnical Commission (IEC) ซึ่งทำหน้าที่ดูแลในเรื่องของข้อกำหนดมาตรฐานและกิจกรรมอวกาศของยุโรป ได้เริ่มให้การแนะนำ (เมื่อราว 5-6 ปีที่แล้ว) สำหรับผู้ที่ต้องการจะเข้ามาประกวดราคาเพื่อออกแบบงานต่าง ๆ ที่เกี่ยวกับซอฟต์แวร์ว่า จะต้องใช้ formal methods จึงจะได้รับการพิจารณา นอกจากนี้ก็มีกระทรวงกลาโหมของอังกฤษ รวมทั้งคณะกรรมการควบคุมด้านพลังงานของประเทศแคนาดาที่บังคับใช้ formal methods ในโครงการซอฟต์แวร์ มาถึงจุดนี้ก็อาจมีผู้สงสัยว่าหน่วยงานและองค์กรต่าง ๆ เท่าที่พูดมานี้ได้ตีความคำว่า formal design กันอย่างไร รวมทั้งปัญหาที่ว่า formal methods จะเข้ามาเพิ่มคุณภาพของซอฟต์แวร์ได้อย่างไร ซึ่งจะอธิบายต่อไป

formal methods กับข้อกำหนดรายละเอียด

ขั้นตอนของการออกแบบซอฟต์แวร์ที่มักนิยมใช้ formal methods ก็คืองานในขั้นสร้างข้อกำหนดรายละเอียด (specification phase) แต่ก่อนที่จะถึงขั้นนั้นก็จะขอพูดถึงการสร้างซอฟต์แวร์ที่ทำกันโดยทั่วไปเสียก่อน กล่าวคือ ก่อนที่งานเขียนคอมพิวเตอร์โปรแกรมจะเริ่มขึ้นนั้น ทีมงานซึ่งประกอบด้วยผู้เกี่ยวข้องต่าง ๆ ก็ต้องมาประชุมเพื่อหารือว่าจะให้โปรแกรมดังกล่าวทำอะไรให้บ้าง อันไม่ว่าจะเป็นไมโครโค้ด (microcode) ที่ควบคุมการเปิดและปิดสวิตช์ต่าง ๆ ในชิป โปรแกรมที่ใช้ในการควบคุมการทำงานของเบรกของรถยนต์ หรือซอฟต์แวร์แพ็กเกจสำหรับงานด้านบัญชี ฯลฯ ขั้นต่อไปน่าจะเป็นเรื่องของการแบ่งแยกงานในหมู่ของผู้เขียนโปรแกรมว่าใครต้องทำอะไรบ้าง รวมทั้งกำหนดแล้วเสร็จและการที่จะต้องโปรแกรมที่รวมเอาข้อต้องการ (specified functions) ต่าง ๆ เข้าไว้โดยไม่มีการตกหล่น ข้อกำหนดรายละเอียดอย่างคร่าว ๆ ในขั้นต้นสำหรับในกรณีนี้จึงมักเขียนขึ้นมาเป็นภาษาชาวบ้าน หรือในรูปของบันทึกการประชุมว่าใครต้องทำอะไรและเสร็จเมื่อไหร่ รวมทั้งข้อต้องการปลีกย่อยอื่น ๆ อันจะตรงข้ามกับสิ่งที่เรียกว่า formal specification ที่ประกอบด้วยกลุ่มของ logic equations ซึ่งตามความเห็นของนายแอนโธนี ฮอลล์ ได้มีว่า การแปลงข้อคิดต่าง ๆ ในเรื่องของการออกแบบซอฟต์แวร์ให้เป็นสมการดังกล่าวเป็นของดีในแง่ของการทำให้เป้าหมายหรือข้อต้องการต่าง ๆ ในการออกแบบมีความชัดเจนและเข้าใจง่าย ดังเช่นการออกแบบซอฟต์แวร์ที่ใช้สำหรับควบคุมการทำงานของเครื่องคิดเงิน ตรงจุดที่เป็นขาออก (electronic checkout machines) ซึ่งถ้าเขียนเป็นแบบ formal specification โดยไม่นำรายละเอียดอื่น ๆ มาพิจารณาให้เป็นที่มากมายสำหรับในขั้นต้น ก็จะได้ว่า

[Scanner (input : COST) (output : DISPLAY)

TOTAL : = TOTAL + COST ;

DISPLAY : = VIEW TOTAL]

ความหมายของสมการในข้างต้นก็คือให้แสดงยอดเงินที่เก็บได้จากการขายที่ประกอบไปด้วยราคาของสินค้าชนิดต่าง ๆ ซึ่งอาศัยสแกนเนอร์เป็นเครื่องมือในการเก็บข้อมูล โดยให้แสดงยอดเงินดังกล่าวบนสกรีนของแคชรีจิสเตอร์

แคชรีจิสเตอร์ในกรณีนี้จึงควรมีปุ่มกดสองปุ่ม นั่นก็คือ ปุ่มหนึ่งสำหรับเปิดลิ้นชัก ส่วนอีกปุ่มสำหรับปิดลิ้นชัก หากจะเขียนข้อกำหนด (สเปก) ในข้างต้นให้ละเอียดขึ้นไปอีกก็จะต้องแจงว่าคอมพิวเตอร์จะเก็บยอดเงินไว้ในเมมโมรีได้อย่างไรในขณะที่กำลังเก็บบันทึก

รายการของสินค้าเมื่อมีผู้มาจ่ายเงิน รวมทั้งจะต้องหาทางทำให้คอมพิวเตอร์สามารถบันทึกได้ว่าลิ้นชักได้เปิดอยู่หรือปิด ตลอดจนการทำงานของสแกนเนอร์ตลอดช่วงเวลาที่เปิดร้านเพื่อจำหน่ายสินค้า อนึ่ง หากจะดูเพียงผิวเผินก็อาจเข้าใจไปว่า formal specification ในข้างต้นเหมือนกับคอมพิวเตอร์โค้ด แต่ความจริงแล้วไม่ใช่ เพราะ input และ output จะหมายถึงสิ่งที่พนักงาน checkout สามารถแลเห็น นอกจากนี้ก็ยังไม่อัลกอริธึมที่บอกว่าคอมพิวเตอร์จะนำตัวเลขต่าง ๆ มาแสดงบนสกรีนได้อย่างไร ฯลฯ จากนี้ก็เป็นหน้าที่ของซอฟต์แวร์โปรแกรมเมอร์ที่จะแปลง logic equations ข้างต้นให้เป็นคอมพิวเตอร์โค้ด สำหรับขั้นต่อไปในการเขียนข้อกำหนดที่เป็นแบบฟอร์มอลให้ละเอียดมากขึ้น เพื่อแสดงการทำหน้าที่ของลิ้นชักของรีจิสเตอร์ในช่วงระยะเวลาต่าง ๆ ของการคิดเงินกับลูกค้ารายใดรายหนึ่งจะเป็นดังนี้

[Drawer (input : OPEN, CLOSE) (output : DRAWER, DISPLAY)

if (OPEN IS PRESSED) and (DRAWER IS CLOSED)

then DRAWER : = OPEN ;

if (CLOSED IS PRESSED) and (DRAWER IS OPEN)

then DRAWER : = CLOSED ; TOTAL : = ZERO ;

DISPLAY : = VIEW ZERO]

ซึ่งหมายความว่าเมื่อมีการกดปุ่ม OPEN หลังจากที่ได้สแกนสินค้าทุกรายการลิ้นชักก็จะเปิดออก สำหรับการปิดลิ้นชักเมื่อกดปุ่ม CLOSE นั้นจะทำให้ตัวเลขต่าง ๆ บนดิสเพลย์กลายเป็นศูนย์เพื่อให้พร้อมที่จะให้บริการแก่ลูกค้ารายถัดไป อย่างไรก็ตาม หากจะพิเคราะห์ให้รายละเอียดก็จะเห็นข้อผิดพลาดที่ว่า ควรจะทำการสแกนรายการของสินค้าต่อไปอีกหรือไม่ ในขณะที่ลิ้นชักยังเปิดอยู่ ซึ่งในทางปฏิบัติถือว่าไม่ควรปล่อยให้ลิ้นชักเปิด อันทำให้ต้องเพิ่มสมการเข้าไปอีกสมการหนึ่งว่า

(SCANNER ENABLED) implies (DRAWER IS CLOSED)

สมการข้างต้นมีชื่อเรียกในทางคณิตศาสตร์ว่า invariant เพราะเป็นคุณสมบัติที่เป็นจริงตลอดเวลาไม่ว่าพนักงาน checkout จะทำอะไรอยู่ จึงเหมาะที่จะใช้ในการตรวจสอบความถูกต้องในการออกแบบซอฟต์แวร์เป็นอย่างยิ่ง เพราะน่าจะทำให้ผู้ออกแบบสามารถเห็นข้อผิดพลาดในขั้นออกแบบและทำการแก้ไขเสียก่อนที่จะมีการเขียนโปรแกรมจริง ๆ ทั้งหมดนี้ก็เป็นแค่เพียงตัวอย่างที่ง่ายที่สุดอันหนึ่งของ formal specification ซึ่งก็ยังมีการมองข้ามในรายละเอียดบางอย่างไปโดยเฉพาะในเรื่องของปฏิสัมพันธ์ระหว่างส่วนต่าง ๆ (interaction between components) ที่ประกอบกันขึ้นเป็นระบบดังเช่น electronic checkout machine ปัญหาจึงมีว่าถ้าเป็นระบบที่มีความซับซ้อนมากกว่านี้นับเป็นพันหรือหมื่นเท่า formal methods จะเข้ามาช่วยได้แค่ไหน ซึ่งก็ต้องไปดูการสำเร็จเพื่อเก็บข้อมูลเชิงกรณีศึกษาที่มีผู้ทำไว้เช่นที่ JPL (Jet Propulsion Lab) ในสหรัฐฯ

การศึกษาของ JPL ได้พบว่า สองในสามของข้อบกพร่อง (defects) ของซอฟต์แวร์ที่ใช้ในกิจกรรมอวกาศ (เช่นการนำร่อง ฯลฯ) เกิดจากการตกหล่นในชั้นของ specification รวมทั้งยังได้พบอีกว่า safety critical faults คิดเป็นจำนวน 197 แห่ง ที่ตรวจพบได้ในยาน Voyager และ Galileo มีเพียงสามแห่งเท่านั้นที่เกิดจากคอมพิวเตอร์โปรแกรม ที่เหลือได้เกิดจากข้อต้องการที่ไม่ได้กำหนดไว้อย่างละเอียดทุกแง่ทุกมุม หรือเกิดจากการเข้าใจในเรื่องของความต้องการของผู้ใช้ที่ผิดไปจากที่ควรจะเป็น ดังนั้นหากจะยังคงใช้ mathematical specification ต่อไปอีกในกิจกรรมขนาดใหญ่เช่นกิจกรรมอวกาศก็จะต้องหันเหความสนใจของทีมงานที่ทำหน้าที่ออกแบบซอฟต์แวร์ให้รู้จักตั้งคำถามด้วยมุมมองอันใหม่ เช่น ผู้ใช้จะใช้ "ระบบ" กันอย่างไรเมื่ออยู่ในสนาม ดังเช่นกรณีของบริษัท Praxis ซึ่งได้ใช้ formal specification ในการแจกแจงรายละเอียดสำหรับงานออกแบบทั้งหมดของระบบควบคุมจราจรทางอวกาศ โดยมองจากในทัศนะของผู้ใช้ การเขียนข้อกำหนดหรือ specification ตามนัยนี้จึงจำกัดอยู่ที่สมการต่าง ๆ ที่ใช้สำหรับแสดงรายละเอียดในกระบวนการ (processes) ว่าจะเกิดอะไรขึ้นเมื่อได้มีการกดปุ่มใดปุ่มหนึ่ง ในกรณีนี้คณิตศาสตร์ได้เข้ามาช่วยผู้ออกแบบซอฟต์แวร์ด้วยการบังคับให้เขาเหล่านั้นรู้จักตั้งคำถามต่าง ๆ ที่เกี่ยวกับปฏิสัมพันธ์ระหว่างผู้ใช้กับระบบที่นำมาใช้งานจริง ๆ (final system) ในสนาม เมื่อต้องการจะแปลงข้อต้องการของผู้ใช้ให้เป็น logic equations สำหรับ formal specification ผลก็คือ ทำให้ได้ระบบที่ประกอบไปด้วยอุปกรณ์อิเล็กทรอนิกส์เป็นจำนวน 150 ชิ้นที่ได้รับการออกแบบ เช่น สำหรับใช้ในการจัดการกับข้อมูลที่เกี่ยวกับการบินขึ้นและลงตลอดจนเรื่องของอินฟอร์เมชันดิสเพลย์ ซอฟต์แวร์ของระบบแบบนี้นอกจากจะประกอบด้วยส่วนต่าง ๆ อย่างมากมายแล้ว ยังประกอบด้วยคอนโทรลโปรแกรมต่าง ๆ เป็นจำนวนมากมาย โดยทุกโปรแกรมต้องทำงานอย่างเข้ากันได้ดี เพื่อให้สามารถนำข้อมูลของการบินขาออกและขาเข้าไปแสดงบนสกรีนตัวเดียวกันได้ ระบบที่ได้ด้วยการออกแบบทางซอฟต์แวร์โดยอาศัยวิธีตั้งคำถามจากทัศนะของผู้ใช้จึงเป็นระบบที่ไว้ใจได้สูง เพราะผู้ออกแบบมีความเข้าใจถึงปัญหาต่าง ๆ อย่างลึกซึ้งมากขึ้น ดังจะเห็นได้จากการเกิด faults ที่ค่อนข้างน้อย คือมีแค่ 150 แห่งเท่านั้น (เมื่อได้มีระบบดังกล่าวออกใช้งานในช่วงปี 1992 ถึงปี 1994 ในประเทศอังกฤษ) เมื่อเทียบกับโครงการอื่น ๆ ที่มีความซับซ้อนที่อยู่ในระดับเดียวกัน

ภยันตรายของ extrapolation และทางแก้ไข

การนำหลักเกณฑ์ทางคณิตศาสตร์มาใช้ในโลกของซอฟต์แวร์โปรแกรมไม่ได้หยุดอยู่แค่การนำ formal methods มาใช้ในขั้นเขียนข้อกำหนดรายละเอียด (specification) แต่ยังได้ก้าวไปถึงการใช้เทคนิคที่เรียกฟอร์มอล

แวลิเดชัน (formal validation) ในการทำให้แน่ใจว่าคอมพิวเตอร์โปรแกรมที่สร้างขึ้นมานั้นสามารถทำหน้าที่ต่าง ๆ ตามที่กำหนดไว้ในสเปก (หรือข้อกำหนด) ได้อย่างแม่นยำ อันเป็นกระบวนการที่อาศัยการพิสูจน์ทางคณิตศาสตร์ (mathematical proof) อันเป็นการพิสูจน์ที่คล้ายกับ mathematical inference เช่นที่ใช้ในวิชาเรขาคณิตที่ว่ามุมทั้งสามของสามเหลี่ยมมีค่าเท่ากับ 180 องศา เนื่องจากภาษาโปรแกรมที่อยู่ในรูปของ formal specification (เช่นภาษา occam) ได้อาศัยรากฐานของคณิตศาสตร์ที่เกี่ยวกับการเปลี่ยนแปลงของสถานะ (เช่น ตำแหน่งของสวิตช์ภายในชิป) เมื่อเป็นเช่นนั้นการค้นพบกฎทางคณิตศาสตร์ที่คอมพิวเตอร์โปรแกรมจะอาศัยเป็นข้ออ้างอิงได้จึงน่าจะเหมือนกับกฎต่าง ๆ ที่ Euclid เป็นผู้คนพบ เพื่อใช้สำหรับพิสูจน์ความจริงทางเรขาคณิต สำหรับระบบที่เป็น safety critical นั้น การพิสูจน์ทางคณิตศาสตร์เป็นทางเดียวที่จะทำให้ระบบดังกล่าวสามารถทำงานได้ตามมาตรฐานสากลด้วยความมั่นใจอันสูง อย่างไรก็ตาม ก็ยังคงมีการสร้างความมั่นใจให้แก่ซอฟต์แวร์ส่วนใหญ่โดยอาศัยการทดสอบแบบทรหด อันเป็นการทดสอบการใช้งานในทุก ๆ สภาวะเสร็จแล้วก็อาศัยการคิดคืบไปข้างหน้า (extrapolate) ในเชิงสมมติว่า ถ้าเกิดมีที่ผิดเพียงแห่งหนึ่ง (a bug) แล้วจะสร้างปัญหาให้มากน้อยแค่ไหน อย่างไรก็ตาม การที่จะหาโอกาสที่จะเกิดความล้มเหลวที่มีเพียง 10^{-9} ในการวิ่งโปรแกรมเพียงหนึ่งชั่วโมงนั้นจะต้องใช้เวลาในการทดสอบนานถึง 114,000 ปี ซึ่งเป็นไปไม่ได้ในทางปฏิบัติ อย่างไรก็ตาม การคิดคืบในเชิงสมมติ (extrapolate) โดยอาศัยผลของการทดสอบเพียงจำนวนหนึ่ง ก็ยังเป็นสิ่งที่มีอันตรายตามความเห็นของนายริค บัทเจอร์ แห่ง Langley Research Center ของ NASA ซึ่งเขาเชื่อว่าโอกาสที่ซอฟต์แวร์จะมีปัญหานั้นยังมีความเป็นไปได้อยู่ตลอดเวลา

ประจักษ์พยานของภยันตรายที่เกิดจากการคิดแบบเข้าข้างตัวเองหรือที่เรียกว่า extrapolation นี้จะเห็นได้จากกรณีของการพบ bug ในเพนเทียมโปรเซสเซอร์ของ Intel ที่เป็นข่าวไปทั่วโลกเมื่อปี 1994 คือก่อนที่จะนำเพนเทียมออกสู่ตลาด Intel ได้ทดสอบผลิตภัณฑ์อย่างทรหดด้วยการวิ่งซอฟต์แวร์แอพพลิเคชันต่าง ๆ บนโปรเซสเซอร์ตัวนี้ พร้อมทั้งการทดสอบในแง่ของการคำนวณที่เป็นตัวเลขนับเป็นล้านครั้งด้วยการใช้ random inputs และแม้ว่าวิศวกรของ Intel จะได้พบร่องรอยของปัญหา (ในขณะที่การออกแบบได้ก้าวไปไกลจนถึงขั้นที่ว่า ถ้าทำการแก้ไขก็จะทำให้โปรเซสเซอร์ตัวนี้ออกสู่ตลาดช้ากว่ากำหนด) ก็ยังมีความมั่นใจในผลิตภัณฑ์ตัวนี้ได้ โดยประมาณการออกมาว่า ปัญหาดังกล่าวมีโอกาสที่จะเกิดเพียงหนึ่งครั้งในทุก ๆ 27,000 ปี จากการ extrapolate ครั้นเมื่อทาง IBM ได้มีโอกาสศึกษาปัญหาของเพนเทียมรุ่นปี 1994 ก็ได้พบว่า สิ่งที่เป็น bug มีโอกาสที่จะโผล่ออกมาให้เห็นได้ในทุก 24 วัน ความยากของเรื่องนี้ก็คือการหาว่าซอฟต์แวร์โปรแกรมโดยทั่วไปจะใช้ไมโครโปรเซสเซอร์ตัวนี้อย่างไร ซึ่งก็ยังมีน้อยคนนักที่จะรู้ สำหรับสาเหตุอันเป็นที่มาของ Pentium-bug ก็คือการเกิดข้อผิดพลาด (error) ในซอฟต์แวร์ที่ทำหน้าที่ควบคุมสวิตชิง (switching) ในชิป อย่างไรก็ตาม ก็ยังไม่มีใครรู้ว่าสิ่งที่เรียกว่า formal validation จะเข้ามาช่วยให้การหาข้อผิดพลาดชนิดนี้ในซอฟต์แวร์ง่ายขึ้นหรือไม่ แต่จากการศึกษาของผู้รู้ในด้านนี้ก็ได้ข้อยุติว่า สิ่งที่เป็นกับเพนเทียมโปรเซสเซอร์ของ Intel เมื่อปี 1994 นั้นสามารถที่จะตรวจพบได้ในขณะออกแบบ ดังเช่นในปี 1990 ซึ่ง Inmos (ซึ่งขณะนี้อยู่ในเครือของ SGS Thomson) ได้ร่วมงานกับหน่วยวิจัยในด้านโปรแกรมมิงของมหาวิทยาลัยอ๊อกฟอร์ด เพื่อพัฒนาอิเล็กทรอนิกส์ที่ทำหน้าที่เฉพาะอย่างเพื่อใช้ในทรานสพิวเตอร์ชิปแบบ T800 ที่ออกแบบไว้สำหรับไว้คิดเลขโดยเฉพาะ งานในขั้นออกแบบของโครงการนี้ได้อาศัย formal methods ซึ่งในระหว่างที่มีการทำ specification ก็ได้พบที่ผิดพลาด (errors) จำนวนหนึ่ง และได้มีการแก้ไขหมดทุกแห่งจนไม่มีปัญหาใดตกค้างอยู่

ในช่วงก่อนปี 1995 ได้มีนักวิจัยผู้มีนามว่า สตีเว่น มิลเลอร์ ซึ่งทำงานอยู่ที่ Rockwell International ในรัฐไอโอว่า พร้อมด้วย Mandayam Srivas ซึ่งเป็นนักวิจัยที่ SRI International อันเป็นบริษัทที่ปรึกษาที่อยู่ในรัฐแคลิฟอร์เนีย ได้แสดงว่าการใช้ formal methods ให้มากขึ้นในงานออกแบบซอฟต์แวร์จะสามารถลดข้อผิดพลาดต่าง ๆ ในการออกแบบโปรเซสเซอร์ได้มาก ซึ่งทั้งสองก็ได้นำผลของการวิจัยในเรื่องของการใช้ formal approach ในการออกแบบและสอบทาน (validating) ความถูกต้องของคอนโทรลซอฟต์แวร์ที่ใช้ใน Rockwell AAMP5 โปรเซสเซอร์ ออกแสดงใน workshop ในปี 1994 ที่เมือง Boca Raton ในรัฐฟลอริด้า ชิปดังกล่าวเป็นส่วนหนึ่งในสายของผลิตภัณฑ์ด้านโปรเซสเซอร์ของ Rockwell ที่ได้มีการนำไปใช้ในส่วนประกอบต่าง ๆ ของ Boing 757 และ Boing 767 โดยใช้กับระบบนักบินกล และใช้กับระบบนำร่องด้วยดาวเทียมที่เรียกว่า Navstar Global Positioning System

จำนวนทรานซิสเตอร์ที่มีอยู่ใน AAMP5 โปรเซสเซอร์ตกประมาณ 500,000 ตัว จึงเป็นโอกาสเหมาะที่จะใช้โปรเซสเซอร์ดังกล่าวสำหรับทดสอบอานุภาพของวิธีการที่เรียกว่า formal validation ซึ่งอยู่ในรูปของ Prototype Verification System หรือ PVS อันเป็นระบบที่พัฒนาโดย SRI เพื่อให้การสนับสนุนในด้าน formal specification และ proof checking โครงการทดสอบดังกล่าวใช้เวลา 20 เดือน รวมทั้งระยะเวลา 5 วันที่ใช้ในการฝึกอบรมวิศวกรของ Rockwell เพื่อให้คุ้นกับ PVS

Prototype Verification System ของ SRI

สิ่งแรกที่วิศวกรของ Rockwell ต้องเรียนรู้ในการชั้น PVS ให้เป็นก็คือการรู้จักใช้ logical language สำหรับการสร้างข้อกำหนดรายละเอียด (specification) รวมทั้งการเรียนรู้ในเรื่องของการพิสูจน์ทางคณิตศาสตร์ (mathematical proofs) โดยอาศัยเครื่องช่วยพิสูจน์แบบอัตโนมัติที่เรียกว่า ปรูฟเช็กเกอร์ (proof checker) สำหรับในการใช้งานนั้นก็จะต้องมีการโปรแกรมกฎต่าง ๆ (mathematical rules of programming) เข้าไปในปรูฟเช็กเกอร์ และเมื่อมีการกดปุ่มปรูฟเช็กเกอร์ก็จะทำหน้าที่เป็นผู้ใช้กฎที่ได้เลือกไว้ อันเป็นงานที่น่าเบื่อ รวมทั้งการที่ผู้ใช้ปรูฟเช็กเกอร์อาจต้องทำการพิสูจน์รายละเอียดบางอย่างที่คนไม่อยากรู้ อย่างไรก็ตาม วิธีการเช่นนี้ก็มีข้อดีในแง่ที่ว่า จะสามารถหาข้อผิดพลาดได้ทั้งหมด หลังจากที่ได้รับการฝึกฝน งานขั้นแรกของวิศวกรก็คือการกำหนดรายละเอียดของภาษาแอสเซมบลี (assembly language) ให้อยู่ในรูปแบบเดียวกันกับที่โปรแกรมเมอร์จะใช้ในการโปรแกรมเข้าไปในโปรเซสเซอร์ที่สร้างเสร็จแล้ว อันหมายถึงการแจงรายละเอียดในรูปของ logical equations ว่าแต่ละคำสั่งที่เป็นภาษาแอสเซมบลีทำอะไรบ้าง เมื่อได้มีการกำหนดรายละเอียดในเรื่องของภาษาโปรแกรมอย่างเป็นที่เรียบร้อย งานขั้นต่อไปก็คือการพิสูจน์ว่ามันจะทำหน้าที่เหมือนกับที่คาดไว้หรือไม่ ภายใต้การควบคุมทางอิเล็กทรอนิกส์ของชิป สิ่งที่ทำให้เกิดความยุ่งยากในกรณีนี้ก็คือว่า ภายในแต่ละอินสตรัคชันที่ง่าย ๆ ของภาษาแอสเซมบลีนั้นสามารถส่งผลให้กับชิปได้อย่างมากมายในรูปของการเปลี่ยนแปลงของสวิตช์ต่าง ๆ รวมทั้งการก่อให้เกิดการแลกเปลี่ยนตัวเลขข้อมูลอันมากจนเกินกว่าที่แอสเซมบลีโปรแกรมเมอร์ต้องการจะรู้ หลักเกณฑ์ในการพิสูจน์ว่าโปรแกรมที่เขียนขึ้นมาด้วยภาษาแอสเซมบลีสามารถทำงานได้ตามที่คาดหวังไว้นั้นจะเหมือนกับเรื่องราวของแคชรีจิสเตอร์ซึ่งได้พูดในตอนต้น เว้นแต่ว่าในกรณีของชิปนี้ได้มีขั้นตอนเล็ก ๆ เป็นจำนวนมากที่เข้ามาเกี่ยวข้อง อันเป็นงานที่เขียนเยิ่นเย้อและน่าเบื่อ ซึ่งเหมาะที่จะให้คอมพิวเตอร์ทำแทน จนทำให้เกิดมีคอมพิวเตอร์โปรแกรมต่าง ๆ เพื่อให้เข้ามาช่วยในเรื่องของการพิสูจน์ดังกล่าว ส่วนสาเหตุที่มิลเลอร์ (Miller) และศริวะ (Srivas) ได้เลือกที่จะใช้วิธีการที่เรียกว่า PVS ก็เพราะเป็นวิธีการที่ประกอบด้วยขั้นตอนต่าง ๆ ที่เรียกว่า inference steps ซึ่งสามารถใช้สำหรับแปลงสิ่งที่ต้องการจะพิสูจน์ (a proof goal) ให้เป็นข้อต้องการที่จะพิสูจน์ที่เป็นหัวข้อย่อย ๆ ที่ง่ายขึ้น (simpler subgoals) ตลอดจนการสามารถที่จะละทิ้งสิ่งที่ไม่ต้องการได้โดยอัตโนมัติ ดังนั้นสิ่งที่วิศวกรทุกคนต้องทำสำหรับในกรณีของ AAMP5 โปรเซสเซอร์ ก็คือการเริ่มต้นด้วย "คำตอบ" จากนั้นก็ตัดสินใจว่าจะมีกฎ (rule) ข้อไหนที่นำมาใช้ได้

การทำงานของ PVS จะเป็นไปตาม "กฎ" (rules) ที่ได้กล่าวในข้างต้นโดยการพิสูจน์ทีละสมการ รวมทั้งการแตกสิ่งที่จะต้องพิสูจน์ (proof) ให้เป็นข้อต้องการที่จะพิสูจน์ที่ย่อยลงไป (smaller proofs หรือ subgoals) เพื่อให้สามารถทำการพิสูจน์ข้อย่อย ๆ ในแบบที่แยกจากกันได้อย่างเป็นอิสระ เช่นในกรณีของการใช้คำสั่งให้มีการคัดลอกข้อมูลจากตำแหน่งหนึ่งไปยังอีกตำแหน่งหนึ่งนั้นวิศวกรจะต้องพิสูจน์ว่าข้อมูลที่อยู่ใน memory locations อื่น ๆ ยังคงเดิม ซึ่งสามารถทำได้ด้วยการพิสูจน์แบบแยกจากกัน (separate proof) เพื่อให้แน่ใจว่าข้อมูลไม่เปลี่ยนแปลงสำหรับในแต่ละ memory location เมื่อขั้นตอนเล็ก ๆ เหล่านี้ได้ผ่านไปจนกระทั่งการพิสูจน์ได้สิ้นสุดลง คือเมื่อ subgoals ทั้งหมดเป็นจริง (self euident) เช่น 1 = 1 วิธีการเช่นนี้จะทำให้วิศวกรสามารถโยงอินสตรัคชันต่าง ๆ ของภาษาแอสเซมบลีเข้ากับสัญญาณอิเล็กทรอนิสก์ของชิปแล้วทำการพิสูจน์รายละเอียดต่าง ๆ ที่มีความสำคัญทางเทคนิค เช่น การสามารถที่จะเรียกใช้สัญญาณที่ถูกต้องเมื่อชิปต้องการสิ่งหนึ่งที่น่าสนใจที่สุดสำหรับโครงการทดสอบนี้ รวมทั้งสาเหตุที่ว่าทำไมจึงควรให้ความสนใจอย่างมากกับ formal specifications ก็คือว่า การอาศัยแค่เพียง specification โดยลำพัง ก็สามารถที่จะเปิดเผยข้อผิดพลาด (errors) ในการออกแบบโปรเซสเซอร์ตัวนี้ (AAMP5) ซึ่งเทคนิคของการออกแบบเท่าที่เป็นอยู่ (traditional design) ไม่สามารถบอกได้ คำว่า formality ในทัศนะของมิลเลอร์และศริวะ จึงเป็นการสนับสนุนให้ผู้ที่ต้องเกี่ยวข้องกับงานด้าน specification ให้รู้จัก "มองไปที่มุมอันซ่อนเร้น" รวมทั้งการคิดถึงเงื่อนไขต่าง ๆ ที่ผิดไปจากปกติ ซึ่งมักจะเป็นต้นเหตุของข้อผิดพลาดต่าง ๆ (errors) อนึ่ง วิศวกรในโครงการทดสอบนี้ได้พบที่ผิดอยู่สองแห่งเมื่องานในขั้น specification สิ้นสุดลง ข้อผิดพลาดทั้งสองแห่งดังกล่าวสามารถที่จะทำให้โปรเซสเซอร์ให้ผลที่ผิดพลาดได้ ที่น่าสนใจก็คือว่า การพบข้อผิดพลาดดังกล่าวได้พบในภายหลังจากที่ได้มีการแก้ไขไว้แล้วตั้งแต่ในตอนต้น ด้วยการสอดแทรกคอมพิวเตอร์โค้ดเข้าไป เพื่อเป็นการแก้ปัญหาเฉพาะหน้า ซึ่งการพบที่ผิดที่ได้ทำในภายหลัง (โดยอาศัย PVS) นั้นได้อาศัยการพิสูจน์เป็นหลัก ซึ่งถ้าหากจะมองย้อนไปในอดีตเมื่อตอนที่ได้มีการออกแบบ AAMP5 ในครั้งแรก (traditional design) จนถึงขั้นตอนของการทดสอบและการประกอบชิปตัวนี้ก็จะเห็นว่าข้อผิดพลาดที่เป็นจุดแรกของชิปตัวนี้ ได้เป็นไปในรูปที่ว่าเป็นการยากที่จะตรวจสอบข้อผิดพลาดแบบนี้ด้วยการทดสอบด้วยวิธีการที่

เป็นแบบมาตรฐานที่มีใช้กันอยู่ ส่วนข้อผิดพลาดจุดที่สองเป็นข้อผิดพลาดที่ไม่มีอะไรเร้นลับมากนัก ซึ่งได้มีการพบเมื่อได้ทำการทดสอบแบบที่ใช้กันอยู่ (traditional testing) แต่ก็เป็นการพบข้อผิดพลาดที่ค่อนข้างล่าช้า นั่นก็คือเป็นการตรวจพบเมื่อได้มีการประกอบ AAMP5 การแกะรอยด้วย PVS จึงไม่เพียงแต่ทำให้วิศวกรตรวจพบข้อผิดพลาดในอดีต แต่ยังทำให้วิศวกรสามารถหาสาเหตุของข้อผิดพลาดได้ทั้งสองแห่งที่ถือว่าเป็นของยากหากจะพึ่งพาอาศัยการทดสอบแต่เพียงอย่างเดียว อันเนื่องจากว่าการทดสอบในครั้งหนึ่ง ๆ นั้นสามารถบอกได้เฉพาะข้อผิดพลาด โดยไม่สามารถบอกสาเหตุ ซึ่งก็จะต้องมีการทดสอบอยู่หลายครั้งเพื่อแยกแยะปัญหา การนำ formal specification มาใช้จึงเกิดอภิประโยชน์ในหลาย ๆ ด้าน นั่นก็คือ การทำให้ผู้ที่ต้องเกี่ยวข้องรับผิดชอบสามารถล่วงรู้ว่ามีปัญหา รวมทั้งรู้สาเหตุแห่งปัญหา ในขั้นตอนที่เรียกว่า proving stage ก่อนที่จะมีการสร้างชิปจริง ๆ ในโรงงาน อย่างไรก็ตาม การที่จะนำสิ่งที่มีคุณประโยชน์อันสูงเช่น formal methods ไปใช้ในวงกว้างในโลกของการพัฒนาซอฟต์แวร์จะต้องคำนึงถึงปัญหาอื่นด้วย

ปรูฟเช็กเกอร์ สำหรับงานพัฒนาซอฟต์แวร์

สาเหตุสำคัญประการหนึ่งที่ทำให้บรรดาผู้พัฒนาซอฟต์แวร์ต้องชะงักเมื่อพูดถึงการนำกรรมวิธีใหม่ ๆ เช่น formal methods มาใช้ในการออกแบบ นั่นก็คือ ค่าใช้จ่ายอันสูงอันเนื่องมาจากความยุ่งยากของวิธีการและความสิ้นเปลืองเวลา ดังจะเห็นได้จากในกรณีของบริษัท Praxis ที่เป็นผู้ออกแบบระบบควบคุมจราจรทางอากาศให้กับ CAA ของอังกฤษ ซึ่งได้แถลงว่าเฉพาะเพียงแค่ขั้น specification ก็หมดเวลาไป 1,500 วัน ซึ่งทั้งมิลเลอร์และศริวะก็เห็นด้วยสำหรับเรื่องนี้ แต่ก็ยังมองไม่เห็นว่าจะมีการตรวจสอบหาข้อผิดพลาดแบบเฉพาะในการออกแบบซอฟต์แวร์สำหรับงานหรือผลิตภัณฑ์ที่มีความอุกฤตสำคัญในแง่ของความปลอดภัยสำหรับคนส่วนใหญ่ (เช่นระบบควบคุมการจราจรทางอากาศ) หรือสามารถส่งผลทางอ้อมแก่ส่วนใหญ่ เช่นการนำ AAMP5 ไปใช้ในโบอิ้ง 757 และโบอิ้ง 767 อันทำให้ต้องมีการพัฒนาสิ่งอำนวยความสะดวกเช่นปรูฟเช็กเกอร์ต่อไปอีก เพื่อให้ทำงานได้อย่างอัตโนมัติ ดังเช่นกรณีของบริษัท Formal Systems (ยุโรป) ที่อยู่ภายใต้การนำของนายไมเคิล โกลด์สมิธ ซึ่งได้พัฒนาปรูฟเช็กเกอร์แบบอัตโนมัติ ซึ่งจะสามารถช่วยให้ผู้ออกแบบในเรื่องของการทำให้การทำ specifications รวมทั้งขั้นของการนำไปปฏิบัติมีความถูกต้องตั้งแต่แรก โดยไม่ต้องลงมือทำทุกอย่างเอง ปรูฟเช็กเกอร์ของนายไมเคิล โกลด์สมิธ มีชื่อเรียกว่า FDR ซึ่งย่อมาจากคำว่า Failures, Divergences, Refinement โดยได้มุ่งการใช้งานไปที่ "ระบบ" ที่ได้รับการออกแบบให้ทำการประมวลผลข้อมูลอย่างขนาน (parallel processing) อันทำให้ได้รับการวิพากษ์วิจารณ์ว่าเป็นการยากอย่างแสนเข็ญที่จะทำให้ระบบดังกล่าวถูกต้อง เพราะจะมีหลายสิ่งหลายอย่างเกิดขึ้นพร้อม ๆ กัน เหมือนกับการทำงานของอุปกรณ์สวิตชิงในชุมสายโทรศัพท์ ซึ่งต้องสามารถจัดการกับกิจกรรมต่าง ๆ อย่างมากมาย ที่สำคัญก็คือการนำแนวความคิดของนายไมเคิลในเรื่องของปรูฟเช็กเกอร์ที่จะใช้กับการออกแบบซอฟต์แวร์ที่เป็นแบบ parallel processing มาแสดงให้อยู่ในรูปของสมการทางคำนวณนั้น นอกจากจะยากมากขึ้นแล้วยังมีความยืดยาวในเรื่องของสัญลักษณ์และตัวเลขทางคำนวณ จนน่าปวดเศียรเวียนเกล้า แต่นายไมเคิลก็มีความเชื่อว่า เครื่องอำนวยความสะดวกที่อยู่ในรูปของ FDR และ PVS สามารถให้ความหวังแก่ผู้ออกแบบที่ไม่ได้เป็นนักคณิตศาสตร์โดยอาชีพ ให้สามารถผนวก formal methods เข้าไปในโครงการการออกแบบซอฟต์แวร์ได้โดยไม่ยาก งานทดสอบเครื่องอำนวยความสะดวกในการออกแบบเช่น FDR และ PVS เพื่อให้กลายเป็นผลิตภัณฑ์ทางการค้าได้เริ่มมาหลายปีแล้วในท่ามกลางความซับซ้อนของซอฟต์แวร์ที่เพิ่มมากขึ้นเรื่อย ๆ (เช่นงานออกแบบโปรเซสเซอร์ชนิดต่าง ๆ จนถึงการออกแบบระบบสาธารณูปโภคที่มีขนาดใหญ่) ปัญหาจึงมีว่า ผู้พัฒนาซอฟต์แวร์เพื่อใช้กับระบบงานที่มีความซับซ้อนมาก ๆ หรือที่เป็นแบบ safety critical จะรอไปได้อีกนานสักเท่าใด โดยไม่ใช้ formal specifications และการพิสูจน์ สุดท้ายของเรื่องนี้จึงต้องขอยืมความเห็นของนายแอนโธนี ฮอลล์ แห่งบริษัท Praxis ที่ว่า "คำถามที่วิศวกรซอฟต์แวร์ควรถามสำหรับในเรื่องของ formal methods จึงไม่อยู่ในประเด็นที่ว่าจะใช้หรือไม่ใช้ แต่จะหาทางกอบโกยผลประโยชน์จากวิธีการดังกล่าวให้มากที่สุดได้อย่างไร เพื่อนำมาทำให้เป็นส่วนหนึ่งของซอฟต์แวร์เอ็นจิเนียริงที่สมบูรณ์แบบ"

คุณภาพของซอฟต์แวร์

ความพยายามที่จะสร้างซอฟต์แวร์ให้มีคุณภาพสูงนั้นน่าจะเป็นเพียงจุดเริ่มต้นของการดำเนินงานเพื่อให้เป็นไปตามแผนที่วางไว้ นั่นก็คือ งานขั้นออกแบบและงานพัฒนาซอฟต์แวร์ จนถึงขั้นทดสอบดังเช่นที่พูดมาแล้ว อย่างไรก็ตาม การที่จะให้ซอฟต์แวร์ที่พัฒนาขึ้นมามีคุณภาพสูงได้ก็จำเป็นที่จะต้องนำกฎเกณฑ์บางอย่างเข้ามาพิจารณา เพื่อให้ซอฟต์แวร์มีความถูกต้องสมบูรณ์ในการใช้งาน ดังนี้

1. ซอฟต์แวร์ที่ดี ควรจะมีความเป็นอิสระในการใช้งานในแง่ของไม่ไปยึดติดกับฮาร์ดแวร์ของระบบคอมพิวเตอร์แบบใดแบบหนึ่งหรือรุ่นใดรุ่นหนึ่ง อันหมายถึงว่าผู้ใช้สามารถที่จะนำซอฟต์แวร์ที่ซื้อมาไปใช้ (execute) กับคอมพิวเตอร์ฮาร์ดแวร์คอนฟิกูเรชันที่กว้างพอสมควร (device-independence)

2. ต้องมีความสมบูรณ์ (completeness) ในแง่ของการที่จะต้องมีส่วนต่าง ๆ ที่จำเป็นอย่างครบครัน รวมทั้งการต้องพัฒนาส่วนต่าง ๆ ดังกล่าวให้สมบูรณ์ (fully developed) คำว่า "จำเป็น" และ "สมบูรณ์" จึงเกินความต้องการในแง่ของหลักวิชาการและการใช้งาน รวมทั้งในเรื่องของดุลยพินิจหรืออาจคลุมถึงจรรยาบรรณความรับผิดชอบในการสร้างซอฟต์แวร์ เว้นเสียแต่ว่าจะเป็นการสร้างซอฟต์แวร์ให้เป็นไปตามข้อต้องการที่ระบุไว้ในสัญญาในรูปของ customized software สำหรับผู้ใช้หรือลูกค้าเฉพาะราย

3. ซอฟต์แวร์ที่ดีต้องมีความแม่นยำ (accuracy) กล่าวคือ ผลที่ได้จากการคำนวณและ/หรือการประมวลผล (outputs) ต้องมีความละเอียดแม่นยำอย่างพอเพียง เพื่อให้สามารถสนองความต้องการในการใช้งานให้ได้อย่างมีประสิทธิภาพและมีความถูกต้อง

4. มีความสมบูรณ์และเป็นหนึ่งเดียวกัน (consistency) ในแง่ของการใช้เครื่องหมายตัวแทน (notation) ศัพท์แสลงหรือนิยาม (terminology) และสัญลักษณ์ (symbology) ในแง่ของการไม่ขัดกันเอง รวมทั้งการมีเนื้อหาของโปรแกรม (หรือของซอฟต์แวร์) ซึ่งสามารถที่จะนำพาไปสู่ข้อต้องการต่าง ๆ ที่ได้กำหนดไว้ว่าซอฟต์แวร์ควรทำอะไรได้บ้าง

5. สามารถทำหน้าที่ต่าง ๆ ที่กำหนดไว้ด้วยการใช้ทรัพยากรชนิดต่าง ๆ อย่างประหยัด (device efficiency)

6. อำนวยความสะดวกในแง่ของการทำให้ผู้ใช้สามารถเลือกใช้ส่วนต่าง ๆ (components) ของซอฟต์แวร์ได้ (accessibility)

7. อำนวยความสะดวกในการป้อนข้อมูล (inputs) และสามารถให้ผล (outputs) ออกมา โดยมีรูปแบบและเนื้อหาที่เข้าใจง่ายและมีประโยชน์ (communicativeness)

8. มีการจัดองค์กรเพื่อเชื่อมโยงส่วนต่าง ๆ ของซอฟต์แวร์เข้าด้วยกันอย่างมีระบบและมีความแน่ชัด (structuredness)

9. มีรายละเอียดภายในซอฟต์แวร์มากพอที่จะให้ผู้อ่านเข้าใจจุดมุ่งหมาย (objectives) สมมติฐาน (assumptions) ข้อจำกัด (constraints) การป้อนข้อมูล (inputs) ผลที่ได้ (outputs) ส่วนต่าง ๆ ที่ประกอบกันเป็นซอฟต์แวร์ (components) และสถานภาพของซอฟต์แวร์ (status)

10. มีความกระชับ (conciseness) ในแง่ของการให้ข้อมูลที่เป็นรายละเอียดต่าง ๆ เฉพาะเท่าที่จำเป็น

11. การทำหน้าที่ในแต่ละส่วนของซอฟต์แวร์ รวมทั้ง statements ที่ใช้ในการบอกส่วนต่าง ๆ ของซอฟต์แวร์จะต้องเห็นได้ชัดจากการอ่านโค้ด (legibility)

12. อำนวยความสะดวกเพื่อให้ผู้ใช้สามารถเพิ่มขีดความสามารถในการใช้งาน (augmentability) เช่นในด้าน data storage ในด้านการคำนวณ (component computational functions)

นอกจากนี้ ก็ยังมีผู้จัดกลุ่มสำหรับคุณสมบัติต่าง ๆ ในข้างต้นออกเป็น 7 กลุ่ม สำหรับในการใช้งานซอฟต์แวร์ (หรือถ้าหากจะมองในทางกลับกัน ก็น่าจะเป็นเครื่องชี้เตือนสำหรับผู้พัฒนาซอฟต์แวร์) ดังนี้

- ความสะดวกในการโอนถ่ายซอฟต์แวร์ไปใช้กับฮาร์ดแวร์แบบต่าง ๆ (portability)
- ไว้ใจได้
- มีประสิทธิภาพ
- คำนึงถึงผู้ใช้ในแง่ของ human engineering
- ทดสอบได้ง่าย
- เข้าใจง่าย
- ปรับปรุงง่าย

ทั้งหมดนี้ก็เป็นข้อต้องการสำหรับการใช้งานซอฟต์แวร์โดยทั่วไป สำหรับในกรณีของซอฟต์แวร์ที่มีความเฉพาะพิเศษซึ่งแตกต่างจากที่พูดในข้างต้น เช่นระบบผู้เชี่ยวชาญ (expert systems) ในทาง AI ก็จะมีคุณสมบัติให้ต้องนำมาพิจารณาที่แตกต่าง (หรือเพิ่มเติม) จากข้างต้น เช่น

- คุณภาพในการช่วยตัดสินและการให้คำแนะนำของระบบ
- ความถูกต้องของ reasoning techniques ที่นำมาใช้
- คุณภาพของการปฏิสัมพันธ์ระหว่างคน (หรือผู้ใช้) กับคอมพิวเตอร์
- ประสิทธิภาพของระบบ และ
- ความคุ้มในแง่ของค่าใช้จ่ายลงทุน (cost effectiveness)

สำหรับเรื่องอื่น ๆ ที่เกี่ยวข้องกับการพัฒนาซอฟต์แวร์ ซึ่งอาจเข้ามาเกี่ยวข้องกับบางโครงการ โดยแยกออกได้เป็นสองประเด็น นั่นก็คือ การพัฒนาแอพพลิเคชันหรือซอฟต์แวร์ในสภาพที่เป็น distributed environment ซึ่งต้องการความร่วมมือจากบุคคลที่กระจายอยู่ในที่ต่าง ๆ ของประเทศ หรือต่างประเทศ และการพัฒนาแอพพลิเคชันหรือซอฟต์แวร์ที่ต้องแบ่งกระจายงานออกไปทำในที่ต่าง ๆ เช่น ออกไปนอกประเทศ ลักษณะของปัญหาสำหรับประเด็นแรกจะเป็นเรื่องของวิศวกรรม ส่วนประเด็นหลังเป็นเรื่องของการบริหาร